Multi-Tenant Data Architecture

June 2006

Frederick Chong, Gianpaolo Carraro, and Roger Wolter
Microsoft Corporation

Applies to:
Application Architecture
Software as a Service (SaaS)

Summary: The second article in our series about designing multi-tenant applications identifies three distinct approaches for creating data architectures. (25 printed pages)

Acknowledgements
Many thanks to Paul Henry for his help with technical writing.

For further reference, an ARCast is available:

ARCast - Software As A Service

Contents

Introduction

Trust, or the lack thereof, is the number one factor blocking the adoption of software as a service (SaaS). A case could be made that data is the most important asset of any business—data about products, customers, employees, suppliers, and more. And data, of course, is at the heart of SaaS. SaaS applications provide customers with centralized, network-based access to data with less overhead than is possible when using a locally-installed application. But in order to take advantage of the benefits of SaaS, an organization must surrender a level of control over its own data, trusting the SaaS vendor to keep it safe and away from prying eyes.

To earn this trust, one of the highest priorities for a prospective SaaS architect is creating a SaaS data architecture that is both robust and secure enough to satisfy tenants or clients who are concerned about surrendering control of vital business data to a third party, while also being efficient and cost-effective to administer and maintain.

This is the second article in our series about designing multi-tenant applications. The first article, Architecture Strategies for Catching the Long Tail, introduced the SaaS model at a high level and discussed its challenges and benefits. It is available on MSDN. Other articles in the series will focus on topics such as workflow and user interface design, overall security, and others.

In this article, we'll look at the continuum between isolated data and shared data, and identify three distinct approaches for creating data architectures that fall at different places along the continuum. Next, we'll explore some of the technical and business factors to consider when deciding which approach to use. Finally, we'll present design patterns for ensuring security, creating an extensible data model, and scaling the data infrastructure.

Three Approaches to Managing Multi-Tenant Data

The distinction between shared data and isolated data isn't binary. Instead, it's more of a continuum, with many variations that are possible between the two extremes.

Data architecture is an area in which the optimal degree of isolation for a SaaS application can vary significantly depending on technical and business considerations. Experienced data architects are used to considering a broad spectrum of choices when designing an architecture to meet a specific set of challenges, and SaaS is certainly no exception. We shall examine three broad approaches, each of which lies at a different location in the continuum between isolation and sharing.

Separate Databases

Storing tenant data in separate databases is the simplest approach to data isolation.

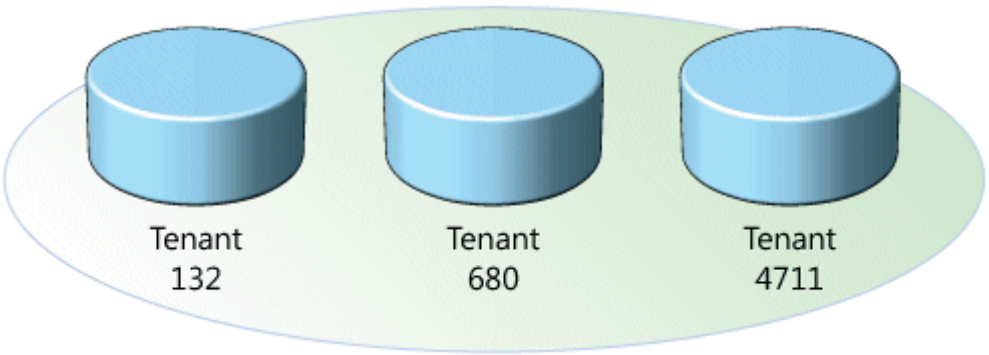

Figure 1. This approach uses a different database for each tenant

Computing resources and application code are generally shared between all the tenants on a server, but each tenant has its own set of data that remains logically isolated from data that belongs to all other tenants. Metadata associates each database with the correct tenant, and database security prevents any tenant from accidentally or maliciously accessing other tenants' data.

Giving each tenant its own database makes it easy to extend the application's data model (discussed later) to meet tenants' individual needs, and restoring a tenant's data from backups in the event of a failure is a relatively simple procedure. Unfortunately, this approach tends to lead to higher costs for maintaining equipment and backing up tenant data. Hardware costs are also higher than they are under alternative approaches, as the number of tenants that can be housed on a given database server is limited by the number of databases that the server can support. (Using autoclose to unload databases from memory when there are no active connections can make an application more scalable by increasing the number of databases each server can support.)

Separating tenant data into individual databases is the "premium" approach, and the relatively high hardware and maintenance requirements and costs make it appropriate for customers that are willing to pay extra for added security and customizability. For example, customers in fields such as banking or medical records management often have very strong data isolation requirements, and may not even consider an application that does not supply each tenant with its own individual database.

Shared Database, Separate Schemas

Another approach involves housing multiple tenants in the same database, with each tenant having its own set of tables that are grouped into a schema created specifically for the tenant.

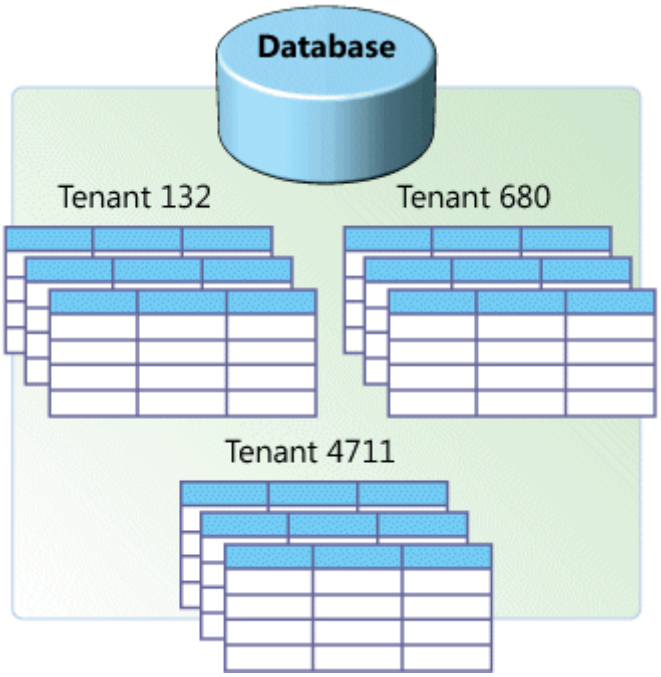

Figure 2. In this approach each tenant has its own separate set of tables in a common database

When a customer first subscribes to the service, the provisioning subsystem creates a discrete set of tables for the tenant and associates it with the tenant's own schema. You can use the SQL CREATE command to create a schema and authorize a user account to access it. For example, in Microsoft SQL Server 2005:

```
CREATE SCHEMA ContosoSchema AUTHORIZATION Contoso
```

The application can then create and access tables within the tenant's schema using the **SchemaName.TableName** convention:

```
CREATE TABLE ContosoSchema.Resumes (EmployeeID int identity primary key,
   Resume nvarchar(MAX))
```

After the schema is created, it is set as the default schema for the tenant account:

```
ALTER USER Contoso WITH DEFAULT_SCHEMA = ContosoSchema
```

A tenant account can access tables within its default schema by specifying just the table name, instead of using the **SchemaName.TableName** convention. This way, a single set of SQL statements can be created for all tenants, which each tenant can use to access its own data:

```
```

```
SELECT * FROM Resumes
```

Like the isolated approach, the separate-schema approach is relatively easy to implement, and tenants can extend the data model as easily as with the separate-database approach. (Tables are created from a standard default set, but once they are created they no longer need to conform to the default set, and tenants may add or modify columns and even tables as desired.) This approach offers a moderate degree of logical data isolation for security-conscious tenants, though not as much as a completely isolated system would, and can support a larger number of tenants per database server.

A significant drawback of the separate-schema approach is that tenant data is harder to restore in the event of a failure. If each tenant has its own database, restoring a single tenant's data means simply restoring the database from the most recent backup. With a separate-schema application, restoring the entire database would mean overwriting the data of every tenant on the same database with backup data, regardless of whether each one has experienced any loss or not. Therefore, to restore a single customer's data, the database administrator may have to restore the database to a temporary server, and then import the customer's tables into the production server—a complicated and potentially time-consuming task.

The separate schema approach is appropriate for applications that use a relatively small number of database tables, on the order of about 100 tables per tenant or fewer. This approach can typically accommodate more tenants per server than the separate-database approach can, so you can offer the application at a lower cost, as long as your customers will accept having their data co-located with that of other tenants.

Shared Database, Shared Schema

A third approach involves using the same database *and* the same set of tables to host multiple tenants' data. A given table can include records from multiple tenants stored in any order; a Tenant ID column associates every record with the appropriate tenant.

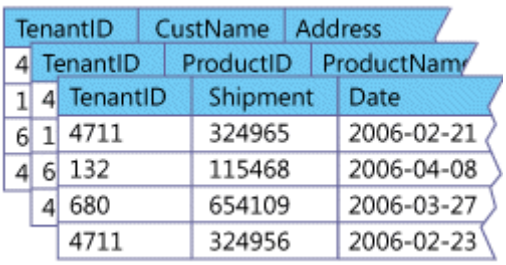

Figure 3. In this approach, all tenants share the same set of tables, and a Tenant ID associates each tenant with the rows that it owns

Of the three approaches explained here, the shared schema approach has the lowest hardware and backup costs, because it allows you to serve the largest number of tenants per database server. However, because multiple tenants share the same database tables, this approach may incur additional development effort in the area of security, to ensure that tenants can never access other tenants' data, even in the event of unexpected bugs or attacks.

The procedure for restoring data for a tenant is similar to that for the shared-schema approach, with the additional complication that individual rows in the production database must be deleted and then reinserted from the temporary database. If there are a very large number of rows in the affected tables, this can cause performance to suffer noticeably for all the tenants that the database serves.

The shared-schema approach is appropriate when it is important that the application be capable of serving a large number of tenants with a small number of servers, and prospective customers are willing to surrender data isolation in exchange for the lower costs that this approach makes possible.

Choosing an Approach

Each of the three approaches described above offers its own set of benefits and tradeoffs that make it an appropriate model to follow in some cases and not in others, as determined by a number of business and technical considerations. Some of these considerations are listed below.

Economic Considerations

Applications optimized for a shared approach tend to require a larger development effort than applications designed using a more isolated approach (because of the relative complexity of developing a shared architecture), resulting in higher initial costs. Because they can support more tenants per server, however, their ongoing operational costs tend to be lower.

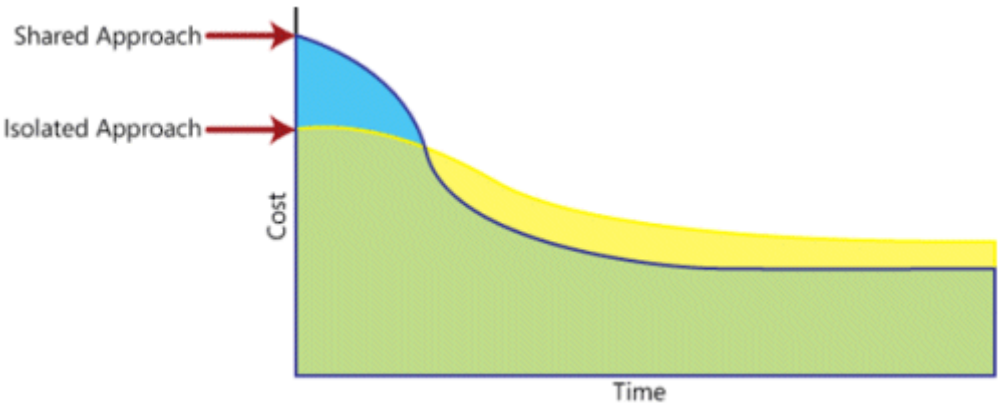

Figure 4. Cost over time for a hypothetical pair of SaaS applications; one uses a more isolated approach, while the other uses a more shared approach

Your development effort can be constrained by business and economic factors, which can influence your choice of approach. The shared schema approach can end up saving you money over the long run, but it does require a larger initial development effort before it can start producing revenue. If you are unable to fund a development effort of the size necessary to build a shared schema application, or if you need to bring your application to market more quickly than a large-scale development effort would allow, you may have to consider a more isolated approach.

Security Considerations

As your application will store sensitive tenant data, prospective customers will have high expectations about security, and your service level agreements (SLAs) will need to provide strong data safety guarantees. A common misconception holds that only physical isolation can provide an appropriate level of security. In fact, data stored using a shared approach can also provide strong data safety, but requires the use of more sophisticated design patterns.

Tenant Considerations

The number, nature, and needs of the tenants you expect to serve all affect your data architecture decision in different ways. Some of the following questions may bias you toward a more isolated approach, while others may bias you toward a more shared approach.

- How many prospective tenants do you expect to target? You may be nowhere near being able to estimate prospective use with authority, but think in terms of orders of magnitude: are you building an application for hundreds of tenants? Thousands? Tens of thousands? More? The larger you expect your tenant base to be, the more likely you will want to consider a more shared approach.
- How much storage space do you expect the average tenant's data to occupy? If you expect some or all tenants to store very large amounts of data, the separate-database approach is probably best. (Indeed, data storage requirements may force you to adopt a separate-database model anyway. If so, it will be much easier to design the application that way from the beginning than to move to a separate-database approach later on.)
- How many concurrent end users do you expect the average tenant to support? The larger the number, the more appropriate a more isolated approach will be to meet end-user requirements.

- Do you expect to offer any per-tenant value-added services, such as per-tenant backup and restore capability? Such services are easier to offer through a more isolated approach.

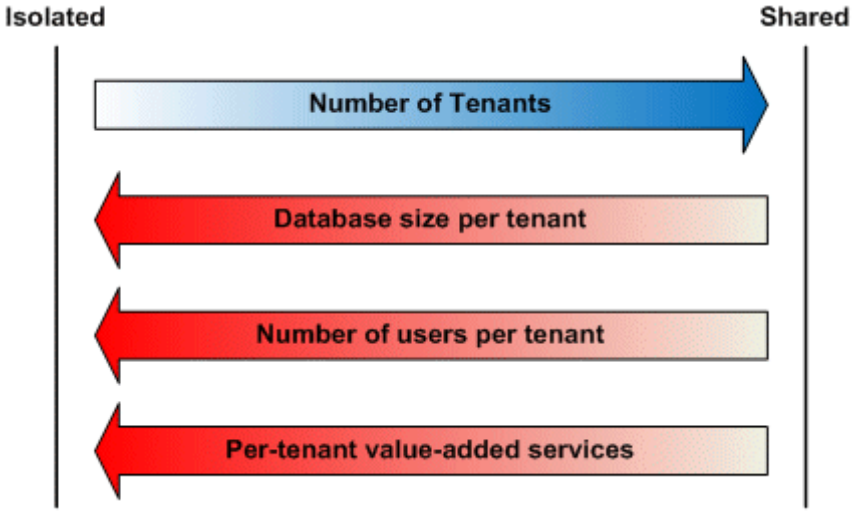

Figure 5. Tenant-related factors and how they affect "isolated versus shared" data architecture decisions

Regulatory Considerations

Companies, organizations, and governments are often subject to regulatory law that can affect their security and record storage needs. Investigate the regulatory environments that your prospective customers occupy in the markets in which you expect to operate, and determine whether they present any considerations that will affect your decision.

Skill Set Considerations

Designing single-instance, multi-tenant architecture is still a very new skill, so subject matter expertise can be hard to come by. If your architects and support staff do not have a great deal of experience building SaaS applications, they will need to acquire the necessary knowledge, or you will have to hire people that already have it. In some cases, a more isolated approach may allow your staff to leverage more of its existing knowledge of traditional software development than a more shared approach would.

Realizing Multi-Tenant Data Architecture

The remainder of this article details a number of patterns that can help you plan and build your SaaS application. As we discussed in our introductory article, a well-designed SaaS application is distinguished by three qualities: *scalability*, *configurability*, and *multi-tenant efficiency*. The table below lists the patterns appropriate for each of the three approaches, divided into sections representing these three qualities.

Optimizing for multi-tenant efficiency in a shared environment must not compromise the level of security safeguarding data access. The security patterns listed below demonstrate how you can design an application with "virtual isolation" through mechanisms such as permissions, SQL views, and encryption.

Configurability allows SaaS tenants to alter the way the application appears and behaves without requiring a separate application instance for each individual tenant. The extensibility patterns describe possible ways you can implement a data model that tenants can extend and configure individually to meet their needs.

The approach you choose for your SaaS application's data architecture will affect the options available to you for scaling it to accommodate more tenants or heavier usage. The scalability patterns address the different challenges posed by scaling shared databases and dedicated databases.

Table 1. Appropriate Patterns for SaaS Application

Approach	Security Patterns	Extensibility Patterns	Scalability Patterns

Separate Databases	• Trusted Database Connections • Secure Database Tables • Tenant Data Encryption	• Custom Columns	• Single Tenant Scaleout
Shared Database, Separate Schemas	• Trusted Database Connections • Secure Database Tables • Tenant Data Encryption	• Custom Columns	• Tenant-Based Horizontal Partitioning
Shared Database, Shared Schema	• Trusted Database Connections • Tenant View Filter • Tenant Data Encryption	• Preallocated Fields • Name-Value Pairs	• Tenant-Based Horizontal Partitioning

Security Patterns

Building adequate security into every aspect of the application is a paramount task for any SaaS architect. Promoting software as a service basically means asking potential customers to relinquish some control of their business data. Depending on the application, this can include extremely sensitive information about finances, trade secrets, employee data, and more. A secure SaaS application is one that provides *defense in depth*, using multiple defense levels that complement one another to provide data protection in different ways, under different circumstances, against both internal and external threats.

Building security into a SaaS application means looking at the application on different levels and thinking about where the risks lie and how to address them. The security patterns discussed in this section rely on three underlying patterns to provide the right kinds of security in the right places:

- **Filtering:** Using an intermediary layer between a tenant and a data source that acts like a sieve, making it appear to the tenant as though its data is the only data in the database.
- **Permissions:** Using access control lists (ACLs) to determine who can access data in the application and what they can do with it.
- **Encryption:** Obscuring every tenant's critical data so that it will remain inaccessible to unauthorized parties even if they come into possession of it.

Keep these patterns in mind as you read the rest of this section.

Trusted Database Connections

In a multi-tier application environment application architects traditionally use two methods to secure access to data stored in databases: *impersonation*, and a *trusted subsystem account*.

With the impersonation access method, the database is set up to allow individual users to access different tables, views, queries, stored procedures, and other database objects. When an end-user performs an action that directly or indirectly requires a call to a database, the application presents itself to the database *as that user*, literally impersonating the user for the purposes of accessing the database. (In technical terms, the application employs the user's *security context*). A mechanism such as Kerberos delegation can be used to allow the application process to connect to the database on behalf of the user.

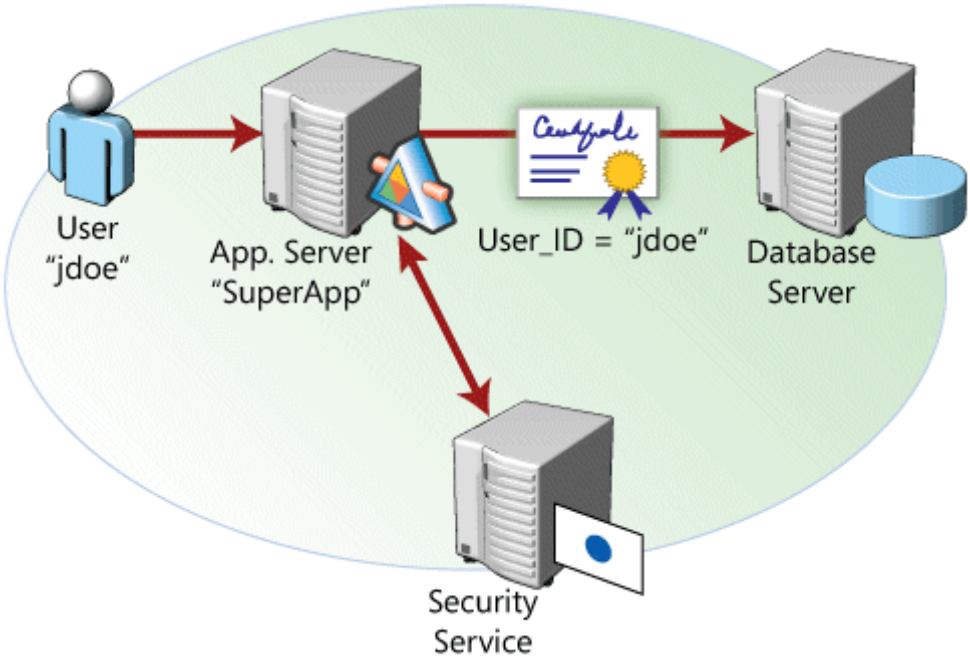

Figure 6. An application connects to a database using impersonation

With the trusted subsystem access method, the application always connects to the database using its own application process identity, independent of the identity of the user; the server then grants the application access to the database objects that the application can read or manipulate. Any additional security must be implemented within the application itself to prevent individual end users from accessing any database objects that should not be exposed to them. This approach makes security management easier, eliminating the need to configure access to database objects on a per-user basis, but it means giving up the ability to secure database objects for individual users.

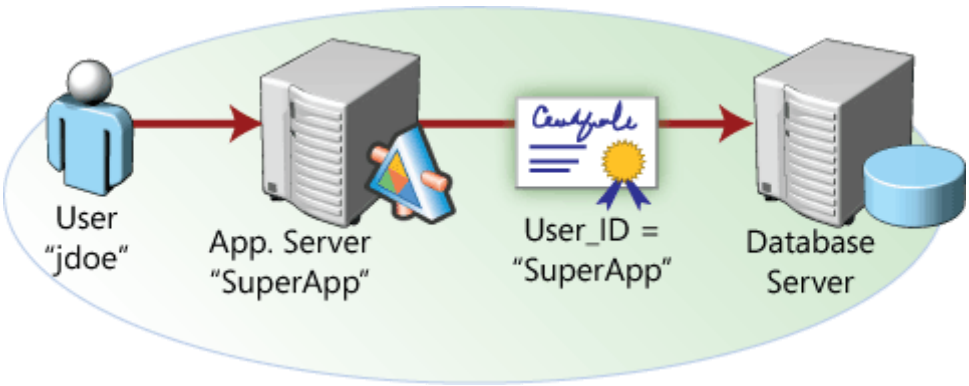

Figure 7. An application connects to a database as a trusted subsystem

In a SaaS application, the concept of "users" is a bit more complicated than in traditional applications, because of the distinction between a tenant and an end user. The tenant is an organization that uses the application to access its own data store, which is logically isolated from data stores belonging to any other tenants. Each tenant grants access to the application to one or more end users, allowing them to access some portion of the tenant's data using end user accounts controlled by the tenant.

In this scenario, you can use a hybrid approach to data access that combines aspects of both the impersonation and trusted subsystem access methods. This allows you to take advantage of the database server's native security mechanisms to enforce the maximum logical isolation of tenant data without creating an unworkably complex security model.

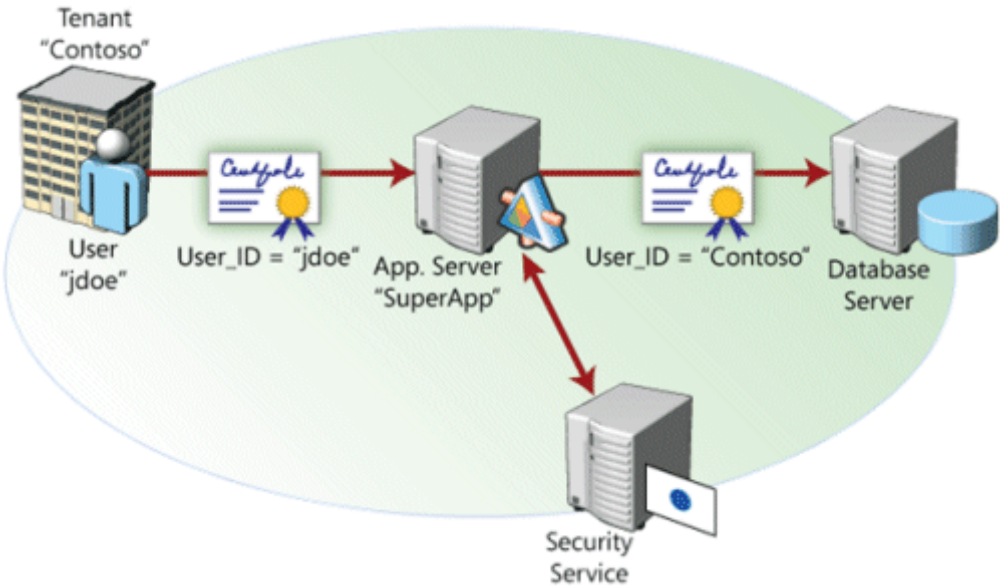

Figure 8. A SaaS application connects to a database using a combination of the impersonation and trusted subsystem approaches

This approach involves creating a database access account for each tenant, and using ACLs to grant each of these tenant accounts access to the database objects the tenant is allowed to use. When an end user performs an action that directly or indirectly requires a call to a database, the application uses credentials associated with the *tenant account*, rather than credentials associated with the end user. (One way for the application to obtain the proper credentials is through impersonation, in conjunction with a credentialing system like Kerberos. A second approach is to use a security token service that returns an actual set of encrypted login credentials established for the tenant, that the application process can then submit to the database.) The database server does not distinguish between requests originating from different end users associated with the same tenant, and grants all such requests access to the tenant's data. Within the application itself, security code prevents end users from receiving and modifying any data that they are not entitled to access.

For example, consider an end user of a customer relations management (CRM) application who performs an operation that queries the database for customer records matching a certain string. The application submits the query to the database using the security context of the tenant, so instead of returning all of the matching records in the database, the query only retrieves the matching rows from the tables the tenant is allowed to access. So far, so good—but suppose the end user's role only allows her to access records of customers located within a certain geographic region. (For more information about roles, see the section "Authorization" in Architecture Strategies for Catching the Long Tail, the first article in this series.) The application must intercept the query results and only present the user with the records that she is entitled to see.

Secure Database Tables

To secure a database on the table level, use SQL's GRANT command to grant a tenant user account access to a table or other database object:

```
GRANT SELECT, UPDATE, INSERT, DELETE ON [TableName] FOR [UserName]
```

This adds the user account to the ACL for the table. If you use the hybrid approach to database access discussed earlier, in which end users are associated with the security contexts of their respective tenants, this only needs to be done once, during the tenant provisioning process; any end user accounts created by the tenant will be able to access the table.

This pattern is appropriate for use with the separate-database and separate-schema approaches. In the separate-database approach, you can isolate data by simply restricting access on a database-wide level to the

tenant associated with that database, although you can also use this pattern on the table level to create another layer of security.

Tenant View Filter

SQL *views* can be used to grant individual tenants access to some of the rows in a given table, while preventing them from accessing other rows.

In SQL, a view is a virtual table defined by the results of a SELECT query. The resulting view can then be queried and used in stored procedures as if it were an actual database table. For example, the following SQL statement creates a view of a table called **Employees**, which has been filtered so that only the rows belonging to a single tenant are visible:

```
CREATE VIEW TenantEmployees AS
SELECT * FROM Employees WHERE TenantID = SUSER_SID()
```

This statement obtains the security identifier (SID) of the user account accessing the database (which, you'll recall, is an account belonging to the *tenant*, not the end user) and uses it to determine which rows should be included in the view. (The example assumes that the unique tenant ID number is identical to the tenant's SID. If this is not the case, one or more additional steps would be required to associate each tenant with the correct rows.) Each individual tenant's data access account would be granted permission to use the **TenantEmployees** view, but granted no permissions to the **Employees** source table itself. You can build queries and shared procedures to take advantage of views, which provides tenants with the appearance of data isolation even within a multi-tenant database.

This pattern is slightly more complex than the Secure Database Tables pattern, but is an appropriate way to secure tenant data in a shared-schema application, in which multiple tenants share the same set of tables.

Tenant Data Encryption

A way to further protect tenant data is by *encrypting* it within the database, so that data will remain secure even if it falls into the wrong hands.

Cryptographic methods are categorized as either *symmetric* or *asymmetric*. In symmetric cryptography, a *key* is generated that is used to encrypt and decrypt data. Data encrypted with a symmetric key can be decrypted with the same key. In asymmetric cryptography (also called public-key cryptography), two keys are used, designated the *public key* and the *private key*. Data that is encrypted with a given public key can only be decrypted with the corresponding private key, and vice versa. Generally, public keys are distributed to any and all parties interested in communicating with the key holder, while private keys are held secure. For example, if Alice wishes to send an encrypted message to Bob, she obtains Bob's public key through some agreed-upon means, and uses it to encrypt the message. The resulting encrypted message, or *cyphertext*, can only be decrypted by someone in possession of Bob's private key (in practice, this should only be Bob). This way, Bob never has to share his private key with Alice. To send a message to Bob using symmetric encryption, Alice would have to send the symmetric key separately—which runs the risk that the key might be intercepted by a third party during transmission.

Public-key cryptography requires significantly more computing power than symmetric cryptography; a strong key pair can take hundreds or even thousands of times as long to encrypt and decrypt data as a symmetric key of similar quality. For SaaS applications in which every piece of stored data is encrypted, the resulting processing overhead can render public-key cryptography infeasible as an overall solution. A better approach is to use a *key wrapping* system that combines the advantages of both systems.

With this approach, three keys are created for each tenant as part of the provisioning process: a symmetric key and an asymmetric key pair consisting of a public key and a private key. The more-efficient symmetric key is used to encrypt the tenant's critical data for storage. To add another layer of security, a public/private key pair is used to encrypt and decrypt the symmetric key, to keep it secure from any potential interlopers.

When an end user logs on, the application uses impersonation to access the database using the tenant's security context, which grants the application process access to the tenant's private key. The application (still impersonating the tenant, of course) can then use the tenant's private key to decrypt the tenant's symmetric key and use it to read and write data.

This is another example of the defense-in-depth principle in action. Accidental or malicious exposure of tenant data to other tenants—a nightmare scenario for the security-conscious SaaS provider—is prevented on multiple levels. The first line of defense, at the database level, prevents end users from accessing the private data of other tenants. If a bug or a virus in the database server were to cause an incorrect row to be delivered to the tenant, the encrypted contents of the row would be useless without access to the tenant's private key.

The importance of encryption increases the closer a SaaS application is to the "shared" end of the isolated/shared continuum. Encryption is especially important in situations involving high-value data or privacy concerns, or when multiple tenants share the same set of database tables.

Because you can't index encrypted columns, selecting which columns of which tables to encrypt involves making a tradeoff between data security and performance. Think about the uses and sensitivity of the various kinds of data in your data model when making decisions about encryption.

Extensibility Patterns

As designed, your application will naturally include a standard database setup, with default tables, fields, queries, and relationships that are appropriate to the nature of your solution. But different organizations have their own unique needs that a rigid, inextensible default data model won't be able to address. For example, one customer of a SaaS job-tracking system might have to store an externally generated classification code string with each record to fully integrate the system with their other processes. A different customer may have no need for a classification string field, but might require support for tracking a category ID number, an integer. Therefore, in many cases you will have to develop and implement a method by which customers can extend your default data model to meet their needs, without affecting the data model that other customers use.

Preallocated Fields

One way to make your data model extensible is to simply create a preset number of custom fields in every table you wish to allow tenants to extend.

TenantID	FirstName	BirthDate	C1	C2	C3
345	Ted	1970-07-02	null	"Paid"	null
777	Kay	1956-09-25	"66046"	null	null
1017	Mary	1962-12-21	null	null	null
345	Ned	1940-03-08	null	"Paid"	null
438	Pat	1952-11-04	null	"San Francisco"	"Yes"

Figure 9. A table with a preset collection of custom fields, labeled C1 through C3

In the previous figure, records from different customers are intermingled in a single table; a tenant ID field associates each record with an individual tenant. In addition to the standard set of fields, a number of custom fields are provided, and each customer can choose what to use these fields for and how data will be collected for them.

What about data types? You could simply choose a common data type for each custom field you create, but customers are likely to find this approach unnecessarily restrictive—what if a customer has a need for three additional string fields and you've only provided one string field, one integer field, and one boolean field? One way to provide this kind of flexibility is to use the string data type for every custom field, and use metadata to track the "real" data type the tenant wishes to use.

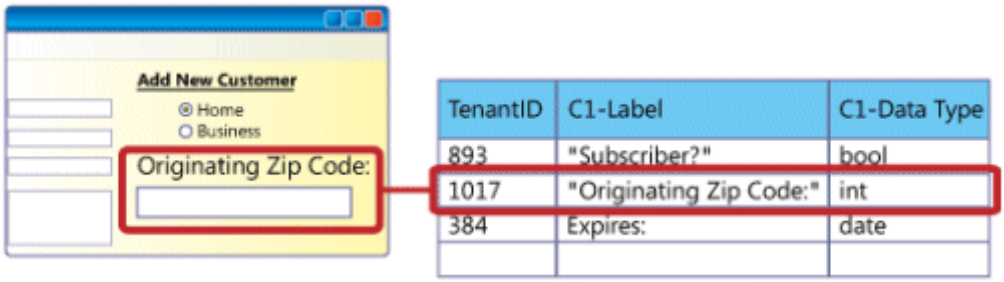

Figure 10. A custom field on a Web page, defined by an entry in a metadata table

In the example above, a tenant has used the application's extensibility features to add a text box called "Originating ZIP Code" to a data entry screen, and mapped the text box to a custom field called C1. When creating the text box, the tenant used validation logic (not shown) to require that the text box contain an integer. As implemented, this custom field is defined by a record in a metadata table that includes the tenant's unique ID number (1017), the label the tenant has chosen for the field ("Originating ZIP Code"), and the data type the tenant wants to use for the field ("int").

You can track field definitions for all of the application's custom fields in a single metadata table, or use a separate table for each custom field; for example, a "C1" table would define custom field C1 for every tenant that uses it, a "C2" table would do the same for custom field C2, and so on.

TableID	C1-Lable	C1-DataType	C2-Lable	C2-DataType
893	"Subscriber?"	bool	"Subscription Code"	string
1017	"Originating Zip Code:"	int	null	null
564	"Expires"	date	"Auto Review?"	bool

Table: *C1ExtensionTable*

TableID	C1-Lable	C1-DataType
893	"Subscriber?"	bool
1017	"Originating Zip Code:"	int
564	"Expires"	date

Table: *C2ExtensionTable*

TableID	C2-Lable	C2-DataType
893	"Subscription Code"	string
564	"Auto Review?"	bool

Figure 11. Storing field definitions in a single metadata table, top, and in separate tables for each custom field

The main advantage of using separate tables is that each field-specific table only contains rows for the tenants that use that field, which saves space in the database. (With the single-table approach, every tenant that uses at least one custom field gets a row in the combined table, with null fields representing available custom fields that the tenant has not used). The downside of using separate tables is that it increases the complexity of custom field operations, requiring you to use SQL `JOIN` statements to survey all of the custom field definitions for a single tenant.

When an end user types a quantity into the field and saves the record, the application casts the value for Originating ZIP Code to a string before creating or updating the record in the database. Whenever the application retrieves the record, it checks the metadata table for the data type to use and casts the value in the custom field back to its original type.

Name-Value Pairs

The Preallocated Fields pattern explained in the previous section is a simple way to provide a mechanism for tenants to extend and customize the application's data model. However, this approach has certain limitations. Deciding how many custom fields to provide in a given table involves making a tradeoff. Too few custom fields, and tenants will feel restricted and limited by the application; too many, and the database becomes sparse and wasteful, with many unused fields. In extreme cases, both can happen, with some tenants under-using the custom fields and others demanding even more.

One way to avoid these limitations is to allow customers to extend the data model arbitrarily, storing custom data in a separate table and using metadata to define labels and data types for each tenant's custom fields.

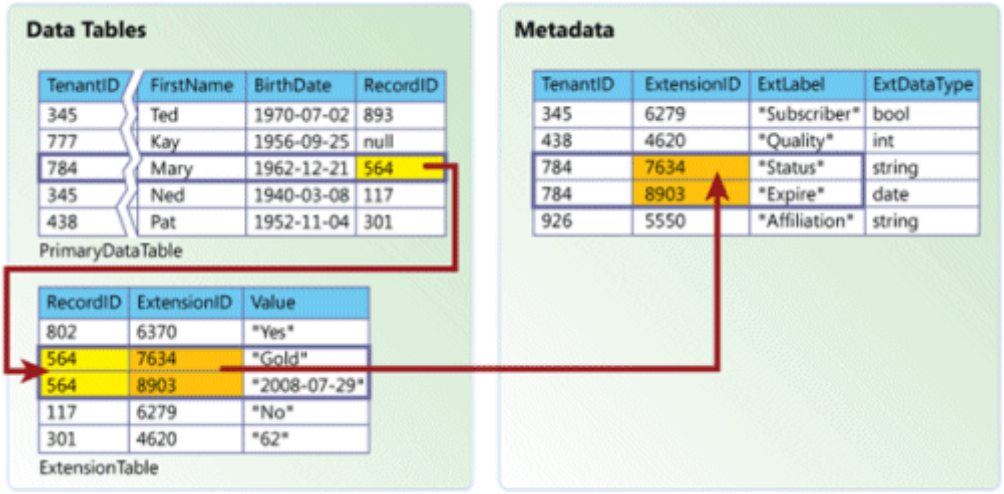

Figure 12. An extension table allows each tenant to define an arbitrary number of custom fields

Here, a metadata table stores important information about every custom field defined by every tenant, including the field's name (label) and data type. When an end user saves a record with a custom field, two things happen. First, the record itself is created or updated in the primary data table; values are saved for all of the predefined fields, but not the custom field. Instead, the application creates a unique identifier for the record and saves it in the Record ID field. Second, a new row is created in the extension table that contains the following pieces of information:

- The ID of the associated record in the primary data table.
- The extension ID associated with the correct custom field definition.
- The value of the custom field in the record that's being saved, cast to a string.

This approach allows each tenant to create as many custom fields as necessary to meet its business needs. When the application retrieves a customer record, it performs a lookup in the extension table, selects all rows corresponding to the record ID, and returns a value for each custom field used. To associate these values with the correct custom fields and cast them to the correct data types, the application looks up the custom field information in metadata using the extension IDs associated with each value from the extension table.

This approach makes the data model arbitrarily extensible while retaining the cost benefits of using a shared database. The main disadvantage of this approach is that it adds a level of complexity for database functions, such as indexing, querying, and updating records. This is typically the best approach to take if you wish to use a shared database, but also anticipate that your customers will require a considerable degree of flexibility to extend the default data model.

Custom Columns

The simplest kind of extensible data model is one in which columns can be added to tenants' tables directly.

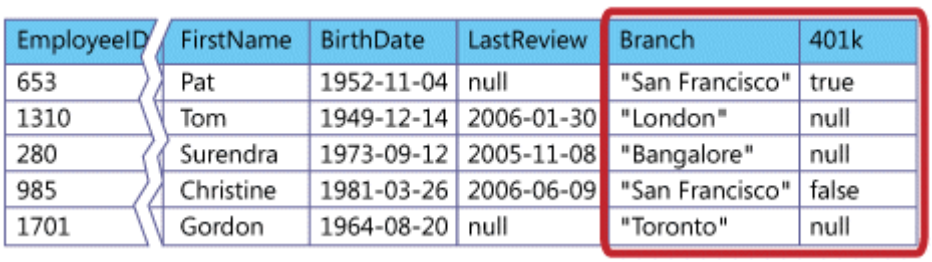

Figure 13. Custom rows can be added to a dedicated table without altering the data model for other tenants

This pattern is appropriate for separate-database or separate-schema applications, because each tenant has its own set of tables that can be modified independently of those belonging to any other clients. From a data model standpoint, this is the simplest of the three extensibility patterns, because it does not require you to track data extensions separately. On the application architecture side, though, this pattern can sometimes be more difficult

to implement, because it allows tenants to vary the number of columns in a table. Even if the Custom Columns pattern is available to you, you may consider using a variation on the Preallocated Fields or Name-Value Pairs pattern to reduce development effort, allowing you to write application code that can assume a known and unchanging number of fields in each table.

Using Data Model Extensions

Whatever method you use to create an extensible data model, it must be paired with a mechanism for integrating the additional fields into the application's functionality. Any custom field implemented by a customer will require a corresponding modification to the business logic (so the application can use the custom data), the presentation logic (so that users have a way to enter the custom data as input and receive it as output), or both. The configuration interface you present to the customer should therefore provide ways to modify all three, preferably in an integrated fashion. (Providing mechanisms through which customers may modify the business logic and user interface will be addressed in a future article in this series.)

Scalability Patterns

Large-scale enterprise software is intended to be used by thousands of people simultaneously. If you have experience building enterprise applications of this sort, you know first-hand the challenges of creating a scalable architecture. For a SaaS application, scalability is even more important, because you'll have to support data belonging to *all* your customers. For independent software vendors (ISVs) accustomed to building on-premise enterprise software, supporting this kind of user base is like moving from the minor leagues to the majors: the rules may be familiar, but the game is played on an entirely different level. Instead of a widely deployed, business-critical enterprise application, you're really building an Internet-scale system that needs to actively support a user base potentially numbering in the millions.

Databases can be scaled up (by moving to a larger server that uses more powerful processors, more memory, and quicker disk drives) and scaled out (by partitioning a database onto multiple servers). Different strategies are appropriate when scaling a shared database versus scaling dedicated databases. (When developing a scaling strategy, it's important to distinguish between scaling your *application* (increasing the total workload the application can accommodate) and scaling your *data* (increasing your capacity for storing and working with data). This article focuses on scaling data specifically.)

Scaling Techniques

The two main tools to use when scaling out a database out are *replication* and *partitioning*. Replication involves copying all or part of a database to another location, and then keeping the copy or copies synchronized with the original. Single master replication, in which only the original (or *replication master*) can be written to, is much easier to manage than multi-master replication, in which some or all of the copies can be written to and some kind of synchronization mechanism is used to reconcile changes between different copies of the data.

Partitioning involves pruning subsets of the data from a database and moving the pruned data to other databases or other tables in the same database. You can partition a database by relocating whole tables, or by splitting one or more tables up into smaller tables *horizontally* or *vertically*. Horizontal partitioning means that the database is divided into two or more smaller databases using the same schema and structure, but with fewer rows in each table. Vertical partitioning means that one or more individual tables are divided into smaller tables with the same number of rows, but with each table containing a subset of the columns from the original. Replication and partitioning are often used in combination with one another when scaling databases.

Tenant-Based Horizontal Partitioning

A shared database should be scaled when it can no longer meet baseline performance metrics, as when too many users are trying to access the database concurrently or the size of the database is causing queries and updates to take too long to execute, or when operational maintenance tasks start to affect data availability.

The simplest way to scaleout a shared database is through horizontal (row-based) partitioning based on tenant ID. SaaS shared databases are well-suited to horizontal partitioning because each tenant has its own set of data, so you can easily target individual tenant data and move it.

However, don't assume, that if you have 100 tenants and want to partition the database five ways, you can simply count off 20 tenants at a time and move them. Different tenants can place radically different demands on an

application, and it's important to plan carefully to avoid simply creating smaller, but still overtaxed, partitions while other partitions go underused.

If you're experiencing application performance problems because too many end users are accessing the database concurrently, consider partitioning the database to equalize the total number of active end-user accounts on each server. For example, if your existing database serves tenants A and B with 600 active users each, and tenants C, D, and E with 400 active users each, you could partition the database by moving tenants C, D, and E to a new server; both databases would then serve 1200 users each.

If you're experiencing problems relating to the size of the database, such as the length of time it takes to perform queries, a more effective partition method might be to target database size instead, assigning tenants to database servers in such a way as to roughly equalize the amount of data on each one.

The partitioning method you choose can have a significant impact on application development. Whichever method you choose, it's important that you can accurately survey and report on whatever metrics you intend to use to make partitioning decisions. Building support for monitoring into your application will help you get an accurate view of your tenants' usage patterns and needs. Also, it's likely that you'll need to repartition your data periodically, as your tenants evolve and change the way they work. Choose a partitioning strategy that you can execute when needed without unduly affecting production systems.

Occasionally, a tenant may have enough users or use enough data to justify moving the tenant to a dedicated database of its own. See the next section, "Single Tenant Scaleout," for help performing further scaling.

The Tenant-based Horizontal Partitioning pattern is appropriate for use with shared-schema applications, which impose some unusual constraints on the familiar task of scaling a database. It provides a way to scale a shared database while avoiding actions that will break the application or harm performance (like, for example, splitting a tenant's data across two or more servers inadvertently or unnecessarily).

Single Tenant Scaleout

If some or all tenants store and use a large amount of data, tenant databases may grow large enough to justify devoting an entire server to a single database that serves a single tenant. The scalability challenges in this scenario are similar to those facing architects of traditional single-tenant applications. With a large database on a dedicated server, scaling up is the easiest way to accommodate continued growth.

If the database continues to grow, eventually it will no longer be cost-effective to move it to a more powerful server, and you will have to *scale out* by partitioning the database on to one or more additional servers. Scaling out a dedicated database is different than scaling out a shared one. With a shared database, the most effective method of scaling involves moving entire sets of tenant data from one database to another, so the nature of the data model that you use isn't particularly relevant. When scaling a database that's dedicated to a single tenant, it becomes necessary to analyze the kinds of data that are being stored to determine the best approach.

The article Scaling Out SQL Server 2005 contains additional guidance and suggestions about analyzing data for scaling out. The article explains reference data, activity data, and resource data in detail, gives some guidelines for replicating and partitioning data, and explains some additional factors that affect scaleout. Some of the scaleout guidelines to consider:

- **Use replication to create read-only copies of data that doesn't change very often.** Some kinds of data rarely or never change after the data is entered, such as part numbers or employee Social Security numbers. Other kinds of data are subject to active change for a defined period of time and then archived, such as purchase orders. These kinds of data are ideal candidates for one-way replication to any databases from which they might be referenced.
- **Location, location, location.** Keep data close to other data that references it. ("Close" in this sense generally means logically proximate rather than physically proximate, although logical proximity often implies physical proximity as well.) Consider the relationships between different kinds of data when deciding whether to separate them, and use replication to distribute read-only copies of reference data among different databases when appropriate.
  For example, if the act of retrieving a customer record routinely involves selecting the customer's recent purchase orders from a different table, try to keep the two tables in the same database, or use replication to create copies of appropriate kinds of data. Try to find natural divisions in the data that will minimize the

amount of cross-database communication that needs to take place. For example, data associated with particular places can often be partitioned geographically.

- **Identify data that shouldn't be partitioned.** Resource data, such as warehouse inventory levels, are usually poor candidates for replication or partitioning. Use scaleout techniques to move other data off the server, leaving your resource data more room to grow. If you have moved all the data you can and still experience problems, consider scaling up to a bigger server for the resource data.
- **Use single-master replication whenever possible.** Synchronizing changes to multiple copies of the same data is difficult, so avoid using multi-master replication if you can. When replicated data must be changed, only allow changes to be written to the master copy.

This pattern can apply to all three approaches, but only comes into play when an individual tenant's data needs cannot be accommodated by a single server. With the separate-database approach, if tenants' data storage needs are modest, each individual server might host dozens of databases; in that case scaling a particular server involves simply moving one or more databases to a new server and modifying the application's metadata to reflect the new data location.

Conclusion

The design approaches and patterns we've discussed in this article should help you create the foundation layer of trust that's vital to the success of your SaaS application. Designing a SaaS data architecture that reconciles the competing benefits and demands of sharing and isolation isn't a trivial task, but these approaches and patterns should help you identify and resolve many of the critical questions you will face. The ideas and recommendations presented here differ in the details, but they all help you leverage the principles of configurability, scalability, and multi-tenant efficiency to design a secure and extensible data architecture for a SaaS application.

This article is by no means the last word in single-instance, multi-tenant data architecture. Later in this series, we'll look at ways you can help tenants put their data model extensions to good use through presentation and workflow customization.

Related Guidance

Developing Multi-tenant Applications for the Cloud on Windows Azure

Feedback

The authors gladly welcome your feedback about this paper. Please email all feedback to fredch@microsoft.com, gianpc@microsoft.com, or rwolter@microsoft.com. Thank you.

© 2016 Microsoft